CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2009)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1482

The impact of IBM Cell technology on the programming paradigm in the context of computer systems for climate and weather models

Shujia Zhou[1,*,†,‡], Daniel Duffy[1], Thomas Clune[1], Max Suarez[1], Samuel Williams[2] and Milton Halem[3]

[1]*NASA Goddard Science Flight Center, Greenbelt, MD 20771, U.S.A.*
[2]*Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720, U.S.A.*
[3]*University of Maryland Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250, U.S.A.*

SUMMARY

The call for ever-increasing model resolutions and physical processes in climate and weather models demands a continual increase in computing power. The IBM Cell processor's order-of-magnitude peak performance increase over conventional processors makes it very attractive to fulfill this requirement. However, the Cell's characteristics, 256 kB local memory per SPE and the new low-level communication mechanism, make it very challenging to port an application. As a trial, we selected the solar radiation component of the NASA GEOS-5 climate model, which: (1) is representative of column-physics components (half of the total computational time), (2) has an extremely high computational intensity: the ratio of computational load to main memory transfers, and (3) exhibits embarrassingly parallel column computations. In this paper, we converted the baseline code (single-precision Fortran) to C and ported it to an IBM BladeCenter QS20. For performance, we manually SIMDize four independent columns and include several unrolling optimizations. Our results show that when compared with the baseline implementation running on one core of Intel's Xeon Woodcrest, Dempsey, and Itanium2, the Cell is approximately $8.8x$, $11.6x$, and $12.8x$ faster, respectively. Our preliminary analysis shows that the Cell can also accelerate the dynamics component ($\sim$25% total computational time). We believe these dramatic performance improvements make the Cell processor very competitive as an accelerator. Copyright © 2009 John Wiley & Sons, Ltd.

*Correspondence to: Shujia Zhou, NASA Goddard Space Flight Center, Greenbelt, MD 20771, U.S.A.
†E-mail: shujia.zhou@nasa.gov
‡Employee of Northrop Grumman Corporation.

Contract/grant sponsor: NASA High End Computing Program
Contract/grant sponsor: ASCR Office; contract/grant number: DE-AC02-05CH11231

Received 26 January 2009; Accepted 23 June 2009

KEY WORDS: IBM Cell processor; climate model; weather model

1. INTRODUCTION

Recent trends in computer microprocessor development have shifted the design point away from improving individual core performance to increasing the number of cores. Depending on the scale, this has been labeled as either multi-core or many-core. This shift implies that in the future, compute-intensive applications, such as Earth and space science models, must exploit orders-of-magnitude additional thread-level parallelism to reach petascale computing and beyond. Higher spatial and temporal resolutions and more sophisticated treatment of physical processes, such as clouds, make models even more computationally intensive. Moreover, the drastic increase in the volume of data collected by various instruments requires a significant increase in computing power for data processing and analysis. Therefore, it is crucial for NASA to both evaluate the impacts of this shift on its computationally intensive modeling and data processing applications and develop appropriate, efficient solutions. One such impact is the memory bandwidth, which often limits computing power of even single-core processors. This problem is exacerbated when adding more cores to the processor, as doing so linearly increases the peak performance resulting in a commensurate linear increase in the requisite memory bandwidth [1].

Recently, several novel architectures have emerged in the commodity processor arena to solve or ameliorate this predicament: IBM's Cell Broadband Engine System (hereafter referred to as 'Cell'), NVIDIA's Tesla General Purpose Graphics Processing Unit (GPGPU), and AMD's FireStream GPGPU. Intel is also developing such a processor, codenamed Larrabee. The Cell processor was chosen for this initial study, since GPGPUs from NVIDIA or AMD were not available when this project started in 2007. Moreover, the fact that Cell supports Fortran as well as double-precision floating-point calculations makes it even more attractive, since most of the Earth and space models are written in Fortran and require double-precision calculations.

The Cell is a multi-core processor offering the capability of increasing performance by one or even two orders-of-magnitude over traditional processors [2,3]. However, the Cell's memory architecture, a 256 kB local memory per synergistic processing element (SPE), coupled with its low-level direct memory access (DMA) communication mechanism makes it very challenging to port a full application such as the NASA Goddard Earth Observing System Model, Version 5 (GEOS-5). Like other climate and weather models, GEOS-5 consists of dynamics (solving Navier–Stokes equations on a thin layer of atmosphere on a global grid) and column physics (parameterizing the sub-grid physical processes such as radiation and clouds). To avoid the complexity of porting a full application in this feasibility study, we selected a single computationally intensive component of GEOS-5 that was well suited to the Cell's characteristics, and effectively exploited the streaming programming model that Cell and GPGPUs support efficiently. In this paper we will describe our approach to porting this solar radiation code to the Cell, present the performance results, and discuss several aspects of the Cell's impacts on supporting climate and weather models from the perspective of a computational center. We will focus on the single-precision, C-version solar radiation code, because the original Cell processor in an IBM BladeCenter QS20 is deficient in its double-precision floating-point performance.

The remainder of the paper is laid out as follows: Section 2 describes the computer processors and compilers used in our performance evaluation; Sections 3 and 4 present analysis of the code and the process of porting it to the Cell; Section 5 shows the performance results; Section 6 discusses the benefits and costs of Cell from the perspective of planning and procuring a computer system. Finally, we draw our conclusions in Section 7.

2. EXPERIMENTAL SETUP

The Cell performance results will be compared against that of the original Fortran component run on three Intel processors at the NASA Center for Computational Sciences (NCCS): Intel Xeon 5150 (Woodcrest) and Intel Xeon 5063 (Dempsey) being used in the Discover computing system, and Intel Itanium2 in the Explore computing system (see Tables I and II). In Table II, Gain of Cell is defined as the performance ratio of Cell with eight SPEs to an Intel processor performance per core. These comparisons are against one core per Intel processor because—unlike with the Cell processor—performance does not linearly scale upwards inside the processor when adding cores, due to the considerable time taken in accessing the data in memory through the front-end memory controller. Since these solar radiation data-columns are independent of each other, no MPI or Cell communication protocol is used to communicate among the cores.

For the Intel processors, the Intel compiler with optimization level -O3 is used. For the Cell processor, both XLC and GCC with -O3 are used for SPEs.

Table I. Characteristics of processors used in the experiment.

Processor	Number of cores	Clock speed	Peak performance per core (single-precision flops)
Cell	1 PPE + 8 SPEs	3.2 GHz for PPE and SPE	8 flops per cycle per SPE
Itanium2	1	1.5 GHz	2 FMA's every cycle (4 flops per cycle per core)
Dempsey	2	3.2 GHz	ADDPS+MULPS every 2 cycles (4 flops per cycle per core)
Woodcrest	2	2.66 GHz	ADDPS+MULPS every cycle (8 flops per cycle per core)

Table II. Peak performance of processors.

Processor	Performance per core	Gain of Cell	Performance per node (single-precision Gflops)
Cell	25.6 (8×3.2)	1.0	$204.8 \times 2 = 409.6$
Itanium2	6.0 (4×1.5)	34.1	$6.0 \times 2 = 12.0$
Dempsey	12.8 (4×3.2)	16.0	$12.8 \times 4 = 51.2$
Woodcrest	21.3 (8×2.66)	9.6	$21.3 \times 4 = 85.3$

3. ANALYSIS OF CODE STRUCTURE AND MEMORY REQUIREMENT

To achieve optimal performance, we must store the data working set as well as the routine's code within the 256 kB local store of each of Cell's SPEs. The small size of the local store places a considerable constraint on the size of data and code working set, which normally uses 1 or 2 GB of main memory in a conventional processor such as Intel's. However, if one had to view the SPE's local store as an L1 cache, rather than the totality of main memory, then 256 kB is quite large.

A typical climate or weather model has a few hundred thousand lines of code and has been developed and written in Fortran over decades. Some portion of the code has been organized with Fortran 90 modules. The code is still evolving with more physical processes (e.g. biochemistry), more detailed physical treatments (e.g. cloud resolving scheme), and newer geophysical grids (e.g. cubed-sphere grid). Considering those constraints and trends, the costs to completely rewrite an application to target-specific architectures are prohibitive. In addition, the modifications to the original code should be minimized, since the researchers need to understand the codes and update them. Finally, any change to the code is acceptable if the performance gains surpass the cost of human effort and computer hardware. Considering all these factors, we chose a solution of accelerating the compute-intensive portion of the code. Since the emerging processors tend to have far larger increases in computing speed than in memory bandwidth and latency, optimal code should have a relatively smaller bandwidth and I/O requirements compared with computation. In addition, the emerging processors tend to only support C when first released. As such, the selected code should have a manageable number of lines of code (e.g. a couple thousand) so that it can easily be converted to C and be modified with hardware-specific libraries (e.g. DMA and SIMD in the case of Cell) to gain maximum performance.

GEOS-5 is written in Fortran and represents main processes in Fortran 90 modules such as finite-volume dynamics core (FVDYCORE) and PHYSICS as shown in Figure 1. SOLAR (the solar radiation component) has been used in various climate and weather models. This component is very computationally intensive, and along with IRRAD (the infrared radiation component) consumes at least 20% of the computing time within GEOS-5. The remaining computing time breaks down as ∼25% for dynamics, ∼25% for input and output data and ∼30% for other column-physics components. The requirements of the solar radiation component in loading and storing data from and to main memory are also small relative to the computational load. These factors make the solar radiation component well suited for one of the Cell's characteristics: high ratio of computation to data transfer capabilities. In addition, this component consists of calculations for each column solely along the vertical direction. As there is no inter-column communication, it is 'embarrassingly' parallel. This characteristic greatly simplifies the coding involving the Cell communication mechanism.

The solar radiation component used in this paper was taken directly from a production version of GEOS-5. Running the solar radiation component in a stand-alone mode required creating a driver (SolarTest()), a stub code (GetAeroIndex()) to limit the number of aerosol types, and the column data file to be processed. Figure 2 shows the detailed code structure along with the memory requirement for input and output variables, temporary (local) variables, and data files. The Cell allows static data such as include (.h) files to be stored into the SPE's local store just like code, without using DMA. Since Cell allows four single-precision floating variables ($4 \times 32 = 128$ bit)

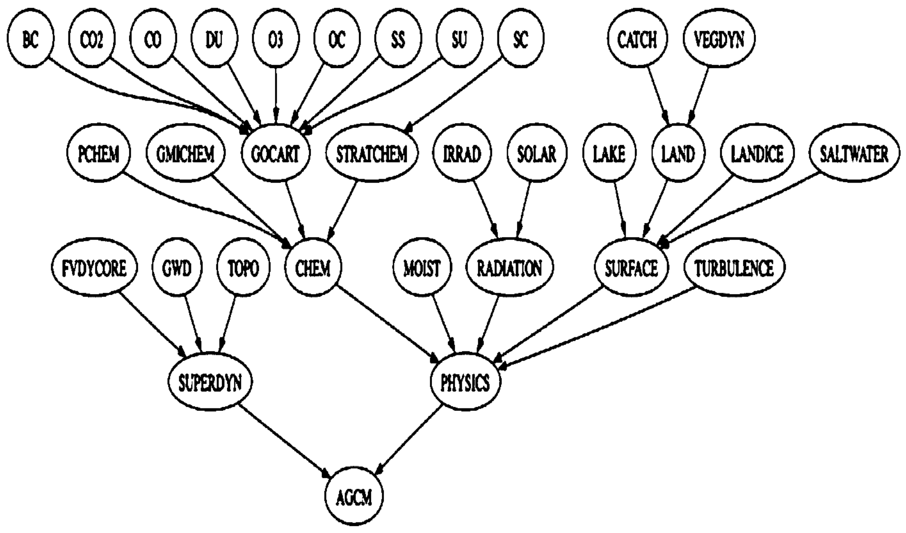

Figure 1. The code structure of the NASA GEOS-5 atmospheric model in terms of components.

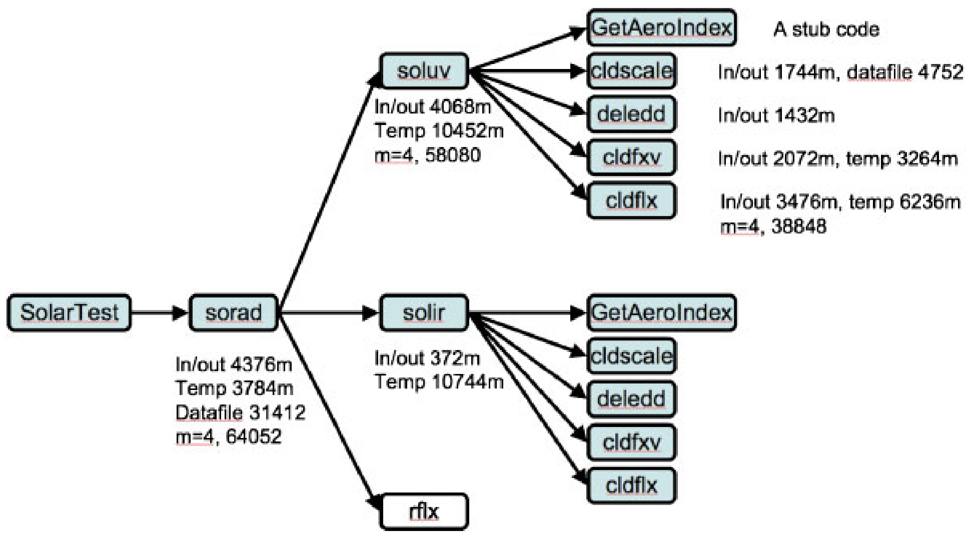

Figure 2. The code structure and memory requirement of the solar radiation model component. In/out, Temp, and Datafile represent the data for input and output variables, temporary variables, the variables in data files. m is the number of data-columns to be processed.

to be processed simultaneously in vector (SIMD) mode, we attempt to take advantage of this vector operation by inputting four columns of data at a time to an SPE. We found that for $m = 4$ (i.e. four columns of data), the data and the associated processing code (sorad() and its called functions) in each SPE is below the 256 kB capacity of an SPE's local store. The maximum size of data involved is ∼161 kB, from sorad(), to soluv(), to cldflx() (see Figure 2).

4. PORTING THE CODE TO CELL

Porting this solar radiation component over to the Cell essentially requires replacing the argument list used in SolarTest() (Figure 2) with DMA calls, as well as setting up the threads for SPEs. Owing to the lack of a Cell SPE Fortran compiler in the summer of 2007, we converted this roughly 2000-line Fortran code to C. Converting the solar radiation core includes three steps: (1) translating the code from Fortran to C, (2) inserting library calls to transfer data between main memory and the SPEs' local store using DMAs, and (3) vectorizing (i.e. SIMDizing) the most computationally intensive function (currently a manual process not performed by the compiler). We would like to point out that the F2C tool [4] is not useful for converting Fortran into C in this situation, since it is very difficult to incorporate the Cell code into the translated version. We found that 16-byte alignment and management of memory addresses (mapping the local multi-dimensional array index in SPEs to the global array index in main memory) required considerable time and attention to implement, since there are 27 various shapes of 1D, 2D, 3D arrays involved in data transfer between PPE and SPEs through DMA. However, we believe that conceptually these modifications are not difficult for a user who knows C and MPI. Steps (1) and (3) took approximately 3 weeks each. However, the newly released IBM XL V11.1 Fortran compiler should obviate the need for steps (1) and (3). At the time this work was conducted, we did not have access to this Fortran compiler and therefore did not evaluate its performance. There is no guarantee that an IBM XL/F compiler will produce high-quality SIMD code. Therefore, a user may still need to explicitly implement step (3) to achieve acceptable performance. Step (2) will not be affected by any foreseeable compilers, and even with a Fortran compiler we would still need to add an additional step of writing a wrapper to use DMA, since its API is in C (there is currently no Fortran binding available).

One major difference between Cell programming and programming in C or Fortran is that the data transfer via DMA is in the style of memory address rather than arrays as illustrated in Figure 3. A user has to convert multidimensional arrays (e.g. 2D and 3D) into 1D arrays. (For the solar radiation code, multidimensional arrays are used even though each column is independent of each other. In a typical array, the first dimension is column index, the second is the number of layers in the vertical direction, and the third is aerosol mixing ratio or effective size of cloud particles.) In addition, he/she needs to decompose the 1D array with a global index into a group of smaller arrays with four array elements since an SPE can only process four columns at once using SIMD (see Figure 4). Consequently a user must convert the main memory global index to an SPE local store index. This is clearly an error-prone procedure. Furthermore, SIMD loads and stores require 16-byte alignment—alien to users in the high-performance computing area.

Our code performance profile shows that most of the computational time is spent on repeatedly executing two functions: deledd() for computing the bulk scattering properties of a single atmospheric layer and cldflx() for computing energy fluxes for the cloud fraction of an atmosphere ranging from 0 to 1. For example, for a calculation running 1024 columns using eight SPEs *without* the optimizations such as vectorization and unrolling, we found that it only takes about 40 µs for one SPE to get data from main memory via DMA. In contrast, it takes approximately 140 000 µs

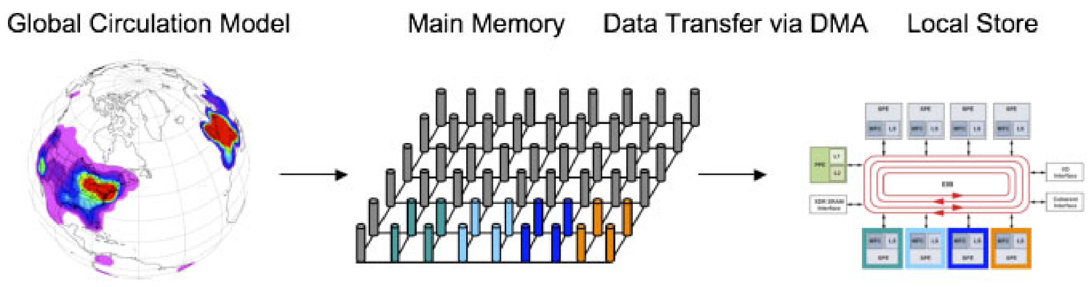

Figure 3. Schematic of mapping the model data from main memory to local stores of SPEs. A global latitude–longitude grid in the horizontal direction and column in the vertical direction are used for a climate model. Columns in groups of four are streamed from main memory into Cell via DMA, processed, and streamed back to main memory.

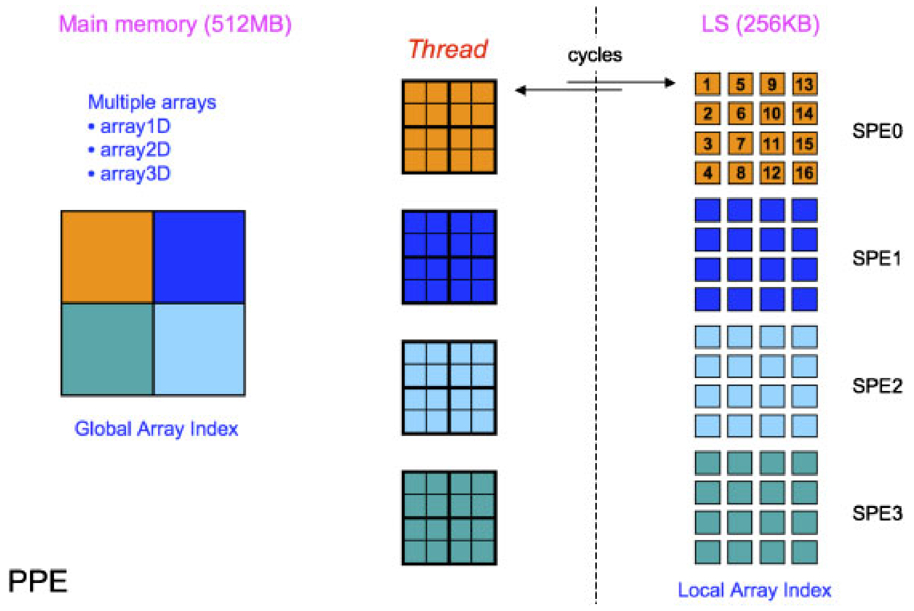

Figure 4. Schematic of data decomposition and transfer via DMA; multiple-dimensional arrays represented in a global array index (on the left side) are decomposed into four segments for four SPEs (on the right side) to process. Each segment is further divided into a group of patches. Those patches are transferred to an SPE to process in a cyclic manner.

(3500 times longer) for one of the major functions, solir(), to run, which calls deledd() 120 times and cldlflx() 30 times. Clearly this is a very computationally intensive calculation. Thus, our effort in using the DMA list, which tends to reduce communication time, did not improve the overall performance noticeably.

5. RESULTS

After manually vectorizing deledd() and cldflx(), we found that one Cell Broadband Engine with eight SPEs in an IBM BladeCenter QS20 can process 3260 columns/s using the GCC compiler with -O3 optimization. Applying unrolling optimizations to these two functions increased the performance by an additional 11%. Furthermore, using the IBM XLC compiler provided an additional gain of roughly 20%.

Figure 5 outlines the performance advantage of the Cell compared with the performance of three different Intel processors running the single-precision version of the code compiled with the Intel Fortran compiler. In comparison with the best baseline results (the single-precision, Fortran-version code with O3 optimization), the Cell achieves a performance speedup of $\sim 8.8x$ over the Intel Xeon Woodcrest, $\sim 11.6x$ over the Intel Xeon Dempsey, and $\sim 12.8x$ over the Intel Itanium2 (see Figure 5 for the comparison of the total number of columns processed for each case).

To better understand the performance improvement across the different processors, we use the single-precision peak computing capability for each type of processor tested. As shown in Table II, the single-precision peak performance gain of the Cell over a single core of the Itanium2, Dempsey, and Woodcrest is $34.1x$, $16.0x$, and $9.6x$, respectively.

Beyond the Cell's powerful computational capability, we believe that there are two essential factors for the dramatic performance improvement over the cache-based Intel processors. First, all computational-support data, such as look-up tables, are stored within the local store of the SPE. Once the data are in the local store, there are no effects from cache and translation lookaside buffer (TLB) misses. Second, explicitly transferring data between the SPE's and main memory via DMA is extremely fast and is a small fraction of the total time.

We note that at least on the conventional processors tested above, the C-version of the code runs measurably slower than the Fortran. For reference, the C-version is slower than the Fortran version

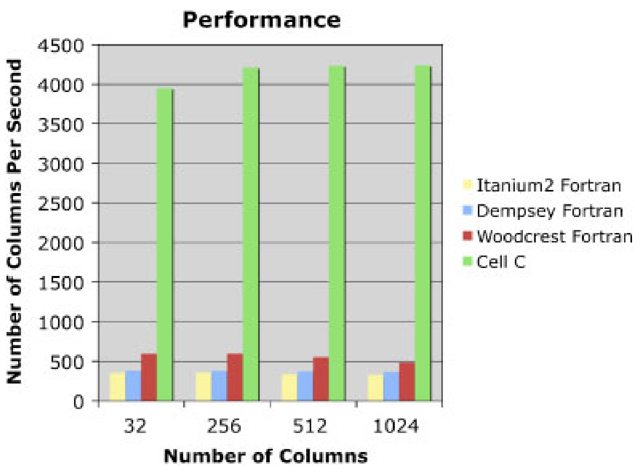

Figure 5. Performance comparison between a Cell processor with eight SPEs and one core of Intel processors, Itanium2, Dempsey, and Woodcrest.

by factors of $1.5x$, $2.6x$ and $3.2x$ on the Woodcrest, Dempsey, and Itanium2, respectively. Hence, the authors expect that the performance numbers stated in this paper should improve when moving to Fortran. Nevertheless, for this paper, we assume that the Cell C performance is not hampered by our language conversion, and we will therefore base our Cell C performance comparisons against the Fortran baseline numbers for the cache-based architectures.

6. DISCUSSION

More than $8x$ improvement on the solar radiation component is certainly very encouraging, since the other column-physics components, such as infrared radiation and moisture, have a similar code structure. That means roughly 50% of the total computational load for the model can expect to obtain significant performance benefits by using the Cell. For the other computationally intensive component, dynamics, our analysis reveals that its most computationally intensive part, solving shallow water equations on a horizontal grid, can be also ported over to the Cell. The approach is to further divide an MPI process domain into a group of patches with self-contained data and process these patches in SPEs in a streaming style, if the data in the MPI process domain are too large for an SPE to process. For example, if we assume (1) five double-precision variables (west-to-east wind, south-to-north wind, temperature, moisture, pressure) on each latitudinal and longitudinal grid point, (2) three ghost cells for stencil calculations in each direction, (3) three time-levels for integration, and (4) 150 kB data allowable in an SPE, then the patch size is 29×29, which is close to the MPI process domain for a $0.5°$ resolution cubed-sphere grid with 256 processors. For finite-volume dynamics with 0.5 degree cubed-sphere grid, the ratio of computation to communication is about 5.0 [5]. So it is expected that Cell technology could significantly improve the dynamics component as well.

While it appears quite likely that most computationally intensive portions of climate and weather models can take advantage of the Cell processor to dramatically improve performance, we must compare these benefits to both the human cost of converting the software and the cost of incorporating Cell nodes within a high-end computing platform.

How can these results be used to architect a cost-effective Cell-based cluster versus a traditional Linux cluster? There are several cost factors that need to be considered, including hardware and software procurement and support, power and cooling, and application porting and tuning for the Cell:

- *Hardware and software procurement and support*: The results above suggest that a single Cell's performance will approximately match two dual-socket, dual-core Intel Woodcrest nodes running the same application with perfect speedup. The more efficient DMA capabilities within a Cell allow applications to run with fewer heavy weight MPI processes thereby reducing the overall communication costs, which in turn results in an increase in the application performance. In addition, this directly correlates to a reduction in the overall communication hardware required to run an equivalent size application in a Cell-based cluster compared with a traditional Linux cluster. It is estimated that the QS20 blade (which has two Cell processors) has the performance equivalent of four Intel Woodcrest nodes, but the cost equivalent of two Intel Woodcrest nodes. This results in a significant hardware and software cost savings of almost 2 to 1 when building equivalent performance clusters for this type of application.

- *Power and cooling*: The cost to power and cool high-performance computing have recently become a very important consideration within the lifecycle cost of a system. As a first-order estimate, a Cell processor consumes less than 100 W while a dual-core Woodcrest consumes approximately 120 W. When adding up all the power required for a node, a Cell node (i.e. a blade with two Cell processors) consumes approximately 210 W while a Woodcrest node will consume roughly 350 W. If a single Cell node is equivalent to four Woodcrest nodes, then the overall power savings will be $4 * 350/210 = 6.6X$. Arguably, this power savings can be doubled when considering the amount of power required to cool the system. Assuming a Woodcrest system with 1024 nodes (∼360 kW) compared with a Cell system with 256 nodes (∼54 kW), this results in a cost savings of over ∼$500 k over a 3-year lifespan of the system.
- *Application porting and tuning*: This is the most difficult cost to estimate as it depends on many factors. In Section 3, we listed our time in porting and optimizing the code, which includes the learning time for Cell programming. However, in the case of an experienced Cell programmer with a C code, it could take 2 to 3 weeks to insert DMA, 2 weeks to SIMDize, and less than 1 week to unroll optimization. In short, the entire conversion of a single component could take 5 to 6 weeks to port and optimize. The most time-consuming part is that of debugging. With the availability of Fortran compilers for the Cell, we expect that the cost of porting codes will be far less in the future. Additional support for OpenMP [6] also has the potential to dramatically reduce the amount of time and effort for porting an application to the Cell. The combined increase in performance and the ease of porting while maintaining portability across multiple architectures makes the Cell platform extremely attractive.

Depending on the application, a cluster of Cell processors or a hybrid cluster of Cell and Linux nodes could result in a better ratio of price/performance than a traditional Linux cluster.

It is interesting to note that one compute-intensive column-physics module, Weather Research and Forecast (WRF) Single Moment 5-tracer (WSM5), representing condensation and precipitation, has recently demonstrated a $20x$ speedup with NVIDIA's GPGPU and CUDA [7]. There is a great interest in comparing CUDA with the Cell programming paradigm. Based on our experience, our preliminary analysis indicates that Cell programming's learning curve is steeper than CUDA. However, Cell programming appears more flexible in supporting applications since SPEs are independent of each other.

7. CONCLUSION

We have found that the IBM Cell technology clearly provides a new way of dramatically improving the performance of climate and weather applications. Identifying the appropriate algorithms is the key to minimizing the porting effort. While the cost of the Cell blade is relatively high compared with traditional Linux nodes, it can potentially be cost-effective when looking at overall application throughput. As the price of the Cell blades continue to drop and their capabilities increase, concurrent with improvements in programmability achieved through the adoption of OpenMP or similar programming models, we expect that hybrid clusters consisting of Cell blades and traditional Linux nodes will become very attractive.

ACKNOWLEDGEMENTS

We would like to thank Carlos Cruz and Bruce Van Aartsen for translating some of the code from Fortran to C. We also would like to thank Tsengdar Lee (NASA High End Computing Program) for providing the funding and Phil Webster for project initiation, Mike Seablom for his inspiration and helpful discussion, John Shalf for sharing his insight on the IBM Cell technology, Lara Clemence for providing assistance in developing Figures 3 and 4, NASA NCCS for installing the IBM Cell Simulator for code development, the Dice Project for training support, and finally, the UMBC Multicore Computational Center for providing access to an IBM BladeCenter QS20 for testing and benchmarking, respectively. Dr Williams was supported by the ASCR Office in the DOE Office of Science under contract number DE-AC02-05CH11231.

REFERENCES

1. Moore SK. Multicore is bad news for supercomputers. *IEEE Spectrum* 2008; 15.
2. Williams S, Shalf J, Oliker L, Kamil S, Husbands P, Yelick K. The potential of the cell processor for scientific computing. *CF'06*, Ischia, Italy, 3–5 May 2006.
3. Scarpazza DP, Villa O, Petrini F. Programming the cell processor. *Dr Dobb's Journal* 2007; 26–31.
4. F2C, a translator from Fortran 77 to ANSI C or C++ developed by Feldman SI, Gay DM, Maimone MW, Schryer NL. Available at: http://www.netlib.org/f2c/src/ [22 July 2009].
5. Putman W. Private communication.
6. O'Brien K, O'Brien K, Sura Z, Chen T, Zhang T. Supporting OpenMP on cell. *International Workshop on OpenMP*, Beijing, 4–5 June 2007.
7. Michalakes J, Vachharajani M. GPU acceleration of numerical weather prediction. *Proceedings of the Workshop on Large Scale Parallel Processing*, *IPDPS*, Miami, 14–18 April 2008. Also available at: http://www.mmm.ucar.edu/wrf/WG2/michalakes_lspp.pdf [22 July 2009].